CS 6373 – Programming Languages, Spring 2014

Programming Assignment #1 – Parsing Configuration Files

Logistics

- This assignment must be done individually. I will run plagiarism detection software over submissions to check to see if anyone is cheating.
- Assignments are due before class on March 21st.
- You are responsible for producing programs that compile and run on my test system, which is Ubuntu 13.10. Generally, as long as you are installing the language implementation I tell you to install, use the standard libraries and use standard Unix Makefiles, you should be okay, even if you do not have an Ubuntu system. But when in doubt, install Ubuntu on a virtual machine and follow the Ubuntu instructions below to match my installation environment.

Overview

The goals of this assignment are to learn about the practical applications of BNF, as well as to explore some historic programming languages. We're going to do this by writing the same tool three different times—a tool to parse very simple configuration files.

All students will have to do an implementation using GNU Flex and Bison, which does require a little bit of C code (it is possible to use other languages with a bit of work, but do not do this—use gcc or the standard CC for your unix-based platform).

- Gnu Bison is available from here: http://www.gnu.org/software/bison/, or on Ubuntu via `apt-get install bison`
- Gnu Flex is available from here: http://flex.sourceforge.net, or on Ubuntu via `apt-get install flex`

Students must do a functional implementation of this project, using only the libraries that come standard with the language. For this, they may choose one of the following languages:

- Scheme, using the GNU (guile) interpreter: http://www.gnu.org/software/guile; available on Ubuntu via `apt-get install guile-2.0`
- ML, using the SML/NJ compiler: http://www.smlnj.org/ ; available on Ubuntu via `apt-get install smlnj`
- Haskell, using the Glasgow Haskell Compiler: https://www.haskell.org/ghc/; available on Ubuntu via `apt-get install ghc`

You may only use a language that you haven't used before taking this class. If you have previously used LISP, you may not use Scheme (they are too similar).

Students must also do an implementation in one of the following additional languages (again, you must use a language that you have never used before taking this class):

- Ada83, using Gnu GNAT: http://www.gnu.org/software/gnat/. Note that your program must compile and run with the –gnat83 compile-time flag (meaning, you must not use any Ada95 features); available on Ubuntu via `apt-get install gnat`
- ALGOL60, using Gnu MARST: ftp://ftp.gnu.org/gnu/marst/marst-2.7.tar.gz ; This is NOT available in any Linux package management systems as far as I know. I just downloaded and followed the instructions.
- Forth, using the pforth implementation: http://www.softsynth.com/pforth/ ; available on Ubuntu via `apt-get install pforth`. Note that Forth is probably the most challenging way to implement this.

For each of these languages, you are expected to be able to install and learn the programming language on your own. Each of these language implementations have documentation freely available on the web-- including tutorials-- with the exception of the ALGOL60 translator, which has documentation in the distribution, in the file: `doc/marst.pdf`.

The Programming Assignment

In this assignment, you will implement three different configuration file parsers to parse hierarchical configuration files intended for an FTP or HTTP server. Here's a simple, well-formed, example configuration file:

```
# This is a comment.
global {
   port = 80
   num_threads    = 4
   ssl_key        = /etc/mycreds.pem
   max_bandwidth = 10.0
};

host www.example.com {
   http_root = /home/example/
};

host www.example2.com {  # This is also a comment.
    http_root = "/home/example\\2/"
    port = 8080
};
```

Valid configuration files

Here’s an informal specification of what constitutes a valid configuration file:

1. Comments may be inserted at any point, except within a string object. They are indicated by a #, and continue to the end of a line.
2. The NULL character is illegal anywhere in the configuration file (ASCII 0x00)
3. The configuration file specifies groups of key/value pairs. The first group of key/value pairs is the global group, which is specified by the keyword “global”. The group must be defined first, although it may have no key/value pairs.
4. Any other group specifies key/value pair overrides for a specific host. Such groups are optional. They are specified using the “host” keyword, followed by a hostname indicator (defined below).
5. The hostname indicator can contain any ASCII alpha-numeric letters, numbers, dots (.), underscores (_) and hyphens (_).
6. Nothing besides white space, the optional terminating semicolon or comments should appear outside a group definition.
7. Group bodies follow the host name or the global key word, and must be surrounded by opening and closing braces. The closing braces may be followed by an optional semicolon.
8. Key/value pairs must not span multiple lines, and must end at a newline, or with the group’s terminating bracket.
9. Key/value pairs are specified by the key, an equals sign, and the value.
10. The key can contain any number of ASCII alphanumeric characters (all standard letters and numbers) and underscores (_), except that the first character must not be a number (the same rules apply to identifiers in C).
11. The value can contain arbitrary signed base 10 integers, arbitrary signed base 10 floats or strings. Only standard representation is acceptable for numbers, not scientific notation.
12. Strings can either be quoted or non-quoted.
13. Non-quoted strings may only contain ASCII-alphanumeric characters, underscores, hyphens (-), the forward slash (/) and the dot (.), and must begin with a letter or a slash.
14. Quoted strings must start with and end with the ASCII double-quote character ("). Any valid ASCII character may appear between the quotes, with the following exceptions:
    a. Newlines and null characters cannot appear.

    b. The backslash (\) indicates the start of an escape sequence: it does not literally translate into the string; it indicates that the next character is to be literally a part of the string (instead of a special character—particularly, a quote).

    c. The double quote (") can only appear within a string if escaped with a backslash (\). This is the primary use of the backslash.

    d. If the backslash is meant to appear in the actual string representation, it may be escaped with a backslash. For instance, "\\hello" in the configuration file, would translate to the string "\hello"

    e. If the backslash appears before the letter \n, then a literal newline is inserted into the string (ASCII hex 0x0a).
    f. If the backslash appears before the letter \r, then a literal carriage return is inserted into the string (ASCII hex 0x0d).
    g. If the backslash appears in front of any other character, the character appears in the string as expected, but the backslash does not.

15. White space always separates lexical tokens, unless inside a quoted string.

Program Input

Here is a specification of how your program should handle input:

1. A single configuration file will be made available in the local directory in which your program runs, in the file “test.cfg”
2. My tests will use no other methods to provide your program with data. If you want to use stdin or command-line arguments for your own purposes (for instance, to turn on debugging), you are welcome to do so, but I will not use any command line arguments in my testing, so your program must work correctly when they are not used.
3. You are expected to process the configuration file in two phases—a lexical analysis phase, where you break up the input into “tokens” into meaningful character strings, and then one where you parse the tokens – analyze them, determining that they are well-formed, and building an appropriate data structure.

Program Output

Here is a specification of what your program should output, if the configuration file is well-formed (i.e., if it is a valid configuration file):

1. It should output “GLOBAL:” followed by a newline, followed by the output specification for key/value pairs (see below).
2. Then, for each host, if any, it should output “HOST <<hostname>>:” followed by a newline, followed by the output specification for key/value pairs, where <<hostname>> should be replaced with the hostname specified in the configuration file, in a case-preserving manner.
3. The output specification for key value pairs is as follows:
    a. For each key/value pair in the configuration group—if any —in order that it appeared in the configuration file, output precisely the following items in this order:
        i. Four spaces.
        ii. A single character indicating the type of the value: “I” for integer, “F” for float, “S” for unquoted string or “Q” for quoted string.
        iii. A single colon.
        iv. If (and only if) the key has previously appeared in this host OR in the global section, the letter “O” (indicating that it’s an override).
        v. A single colon.
        vi. The name of the key, as it appeared in the configuration file

    vii. A single colon.
    viii. If the value is an integer, float or unquoted string, the value as it appeared in the configuration file.
    ix. If the value is a quoted string, print three quotes, then the “unquoted” value, then three quotes. The “unquoted” value is not the literal from the configuration file, but the processed version of the value. For instance, if there was a \\ in the configuration file, then you should output only a single backslash.
    x. A newline
  b. Output a newline (whether or not there were key/value pairs)

Here’s example output for the configuration file above:

```
GLOBAL:
    I::port:80
    I::num_threads:4
    S::ssl_key:/etc/mycreds.pem
    F::max_bandwidth:10.0
HOST www.example.com:
    S::http_root:/home/example
HOST www.example2.com:
    Q::http_root:"/home/example\2/"
    I:O:port:8080
```

Note that , in properly formatted output, nothing appears before GLOBAL, and after 8080, only a single newline appears.

If the program is not well-formed, then use the following rules for output:

1. If the configuration file has an I/O problem (e.g., does not exist), then your program should ONLY output the following:
   a. The word “ERR”,
   b. A single colon
   c. The letter “F” (indicating an error with the file)
   d. A single colon
   e. A single newline
2. If the error occurs during the lexical analysis phase, then your program should ONLY output the following:
   a. The word “ERR”,
   b. A single colon
   c. The letter “L” (indicating the error was in lexical analysis)
   d. A single colon
   e. The line number on which the bad token occurs on (counting from 1)
   f. A single newline
3. If the error occurs during the parsing phase, then your program should do the following, in order:

a. Produce correct output for any groups that were FULLY accepted up until the error (if any), but no output for the group where the error was noticed.
b. The word “ERR”
c. A single colon
d. The letter “P” (indicating the error was in parsing)
e. A single colon
f. The line number on which the error is detected (again, counting from 1)
g. A single newline

Program Compilation and Submission

1. By the end of Feb 28th, I will provide a ZIP file containing three test cases that your parser should accept and three that it shouldn’t. But you should do much more extensive testing based on this specification.
2. You must provide all source code in a form that’s buildable on my machine.
3. All files submitted must be compressed into a single archive file (any standard is fine— for instance, tar.gz, .zip)
4. When I uncompress your submission, all the files in it must uncompress into the directory in which I unzip it.
5. Name your compressed file after your netID. For instance, if I were submitting a ZIP file, I would submit jtv202.tar.gz
6. Do not password protect / encrypt the archive file.
7. You must use a standard Makefile to specify how to build your projects.
8. You must have a target called “all” that is the first target, and builds all three of your programs.
9. For the flex/bison/C program, the makefile must output a single executable called imp1 (with no file extension).
10. If you do the scheme implementation, my test suite will load and run the file imp2.scm (do not pre-compile your program for me).
11. If you do the ML implementation, my test suite will load and run the file imp2.sml (do not pre-compile your program for me).
12. If you do the Haskell implementation, your makefile must output a single executable called imp2h (with no file extension).
13. If you do the Ada83 implementation, your makefile must output a single executable called imp3ada (with no file extension).
14. If you do the ALGOL60 implementation, your makefile must output a single executable called imp3algol (with no file extension).
15. If you do the Forth implementation, my test suite will load and run the file imp3.f.
16. You should not use any third-party libraries or additional languages beyond what I’ve specified. If you think you need to do this, please contact me to discuss and get approval before proceeding.
17. Submit assignments to me via email: viega@nyu.edu, with the subject: Programming Assignment 1

Grading

1/3 of the grade will be for your flex/bison/C implementation.
1/3 of the grade will be for your functional implementation.
1/3 of the grade will be for your additional implementation.

I will grade assignments through an automated program. It will do the following:

1. Unzip your submission.
2. Run make in the directory where all the files unzip.
3. For each test case, run each of your implementations as described above, and compare the output to the correct output.

Your grade for each implementation will be the percentage of test cases you get correct.

Since all the grading is automated, it is VERY important that you follow the specification precisely, so that your programs work in my environment, and that you don't get dinged over errors that you could have easily avoided (such as adding additional spaces or newlines). I will not be flexible at all on these items—your grade is your grade.

The reason I'm being this draconian on the grading is that it's good training for careers in software engineering. Once there's been a decision on what to build, it's your responsibility to get the details right, even if they are small. And if there are ambiguities, it's your responsibility to root them out and question them.

While there is no extra credit for doing extra implementations, if you do extra implementations, I will count the one in each category where you score the best That is, if you do multiple languages from the functional category, I'll count your best functional implementation for that part of your grade; and the same goes for the "other" category.

Final Notes

I recommend you take the following approach to this assignment:

1. Figure out what token types you need to have to represent all the elements in the configuration file language. One thing you need to figure out is how to specify tokens so that you can handle tokens with overlapping ranges unambiguously in your bison grammar.
2. Write the Flex tokenizer.
3. Write the Bison parser, which will give you a formal description of the language (similar to BNF) and the necessary C code to get example 1 to work properly.
4. Once you have the formal description and your first implementation, do as much testing as you can.
5. At this point, having the formal specification and a working implementation should make it a lot easier to do the other two implementations.

Many students will have to teach themselves a lot of technologies for this assignment: Flex, Bison, two to three new programming languages (if you don't know C), and possibly Makefiles. All of these things are well-documented. For example, there are many Makefile tutorials, including one here: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

If you think you need help with these technologies, you can ask very directed questions on the technology at StackOverflow. You can help each other getting technologies running, but you cannot help with the implementation of the assignment.

Finally, if you think there is an ambiguity in this specification, do not make any assumptions. Instead, ask the instructor for clarification.